Greg Chan

EECS 349 Homework 5

Problem 1

Part A

The performance of AdaBoost would be negatively effected by a data set that changes from round to round. This is because AdaBoost assumes a fixed set of data, and the chance of changing will cause the first classifier to be less reliable than the most recently trained classifier. This is because the chance of changing data makes the reliability of this learner less reliable because the data set maybe have changed significantly since it was trained. This negative effect would also be magnified if the weight for that learner is higher than others.

Part B

To improve AdaBoost to work with this changing data set, you could add or modify the existing weight for the with classifier so that the with classifier is not as trusted as the jth where $i < j$. More specifically, the weight of a learner classified more recently will be much higher than one that was trained first. In this way you would put more trust in the classifier that has most recently been trained over one trained earlier.

A way to mathematically do this would be to have a term with the $w_i=\log_2(i/n + 1)$ where i is the index of the classifier and n is the number of learners trained so far. This would mean that the value of this mathematical weight range from 0 to 1 and as the number of classifiers increases and the index for that classifier increases, the value placed on older classifiers would be logarithmically less significant.

Part C

This would be better because if the data changes, we put more trust in the classifier that most recently trained on the data. We discussed this in class… Classifiers that work on one dimension as the example. So if you have a few points and then they change, the classifier at the end will likely work better than the one before it…

Problem 2

Part A

Active learning is a method of semi supervised learning where the learner is able to query the information source to obtain desired outputs. Active learning differs from supervised learning in that it doesn't assume a large set of labeled data from which we can base our model off of. Instead, we are presented with a query and classify method. Essentially we train on a few of the unlabeled data points but because this process is expensive we train on very few points. Then, we build a model around these labeled points. Next, we choose from the pool of unlabeled data that we have and determine the best unlabeled point to get the label for. Once we have the label for this point, we reclassify. What also makes active learning different from passive learning is active learners try to find the points that will best improve the classifier by searching through the space of unlabeled points.

An application of active learning that we discussed in class is the research projects that Professor Pardo and the TAs are working on to put names to parameterizations on equalizers and reverberation parameters in music. Because it's expensive to get this data from users (as in it's potentially time consuming) active learning is used to try to learn the labels as best as

possible in as short of a time as possible. Professor Pardo showed us a simulation of describing bright and not bright sounds when presented with a series of unlabeled parameterizations for equalization. Using the interface, the computer was able to actively learn the difference in parameters between dark and bright sounding sounds.

Part B
Let S = set of specific examples, and G = set of general examples. If we have a specific set of concepts and a general set of concepts like in candidate elimination, we can narrow our version space through queries that fall in the “difference” of S and G.

By examining instances that are in the difference of S and G, if an instance in this region proves positive, then some s in S will have to generalize to accommodate the new information; if it proves negative, some g in G will have to be modified to exclude it. In either case, the version space is reduced with every query. This is like query by committee, which basically says maintain a version space of hypotheses, and pick the instances generating the most disagreement among hypotheses.

Part C
For an SG neural network on the 25bit threshold problem, the function governing the relationship with the number of samples and error rate the learner is an exponential function as show in the graph in the problem in the paper section 4.2 figure 12. When we used random samples then the relationship changes and appears to be polynomial. The sampled networks exhibit a steeper drop in generalization error. This is to be expected from an active learning algorithm when compared to a random samples algorithm.

Problem 3
I coded up this problem in python. My code can be seen in the file ‘markov.py’. Below are my results from the two observation sequences and the two different models, totaling in four different tests. I used the forward algorithm to formulate my results for this question. This algorithm uses the starting probabilities, the transition matrix and the observation matrix to produce a trellis of probabilities at each stage in the sequence.

Model 1, Sequence 1
T=0, probability: [0.3 0.3 0.4]
T=1, probability: [0.03 0.15 0.08]
T=2, probability: [0.013   0.026   0.0156]
T=3, probability: [0.00273   0.00546   0.003276]
T=4, probability: [0.0022932   0.00045864 0.00206388]
T=5, probability: [0.000963144   0.0001926288 0.0008668296]
Probability of observing sequence, given the model: 0.002023

Model 1, Sequence 2
T=0, probability: [0.3 0.3 0.4]
T=1, probability: [0.15 0.09 0.08]
T=2, probability: [0.016   0.032   0.0192]
T=3, probability: [0.01344   0.002688 0.012096]
T=4, probability: [0.0014112   0.0028224   0.00169344]
T=5, probability: [0.000296352   0.000592704   0.0003556224]
Probability of observing sequence, given the model: 0.001245

Model 2, Sequence 1
T=0, probability: [0.5 0.3 0.2]
T=1, probability: [0.05 0.18 0.04]
T=2, probability: [0.0142 0.0378 0.013]
T=3, probability: [0.003332 0.009432 0.003192]
T=4, probability: [0.00247224 0.00038436 0.00271012]
T=5, probability: [0.0007053408 0.000163158   0.0011088028]
Probability of observing sequence, given the model: 0.001977

Model 2, Sequence 2
T=0, probability: [0.5 0.3 0.2]
T=1, probability: [0.3   0.09 0.02]
T=2, probability: [0.0126 0.0684 0.034]
T=3, probability: [0.019188 0.002766 0.016366]
T=4, probability: [0.00153168 0.00673164 0.00235676]
T=5, probability: [0.0005759376 0.001507716   0.0004695688]
Probability of observing sequence, given the model: 0.00255

Based on the probability of observing the sequence, given the two models, it seems clear that Model 2 was better than Model 1 because in both sequences, it was able to produce a higher probability of these sequence of events occurring. Model 2 out performed Model 1 because the average probability of observing the sequence given the different models was higher for Model 2. While Model 2 seems to outperform Model 1, it is still also clear that neither of these models for this series of observations produces a very likely result given that the probabilities are below 0.5% in all cases.

Problem 4
Part A
The first method for modeling state duration for HMM is a model in which observations are associated with the arcs of the model This model allows transition from one state to another with no output. These transitions are called null transitions. There is another variation of the HMM structure which is parameter tying. The idea is to create an equivalence relationship between HMM parameters in different states. Thus the number of independent parameters reduces and its parameter estimation becomes simpler. This technique is most useful where there is insufficient training data to estimate a large number of model parameters.

The second approach to modeling state duration for HMM is to tie the model parameters to reduce the number of parameters. This also simplifies the parameter estimation process by simply removing parameters. This has a very high cost of including duration density with computation cost on the order of D^2 and D fold increase in storage. The state duration probability will be measured directly from the K-means producer, thus the estimates pi(d) are strictly heuristic ones.

Part B
The traditional method used for speech recognition is by use HMM. HMM are prominent in large vocabulary speech recognition. An HMM is a generative model where the observable acoustic features are assumed to be generated from a HMM process that transitions between states.

Deep belief networks are probabilistic generative models with multiple layers of stochastic hidden units above a single bottom layer of observed variables that represent a data vector. DBNs have undirected connection between the top two layers and direction connections to all other layers.

The combination of artificial neural networks and HMMs as an alternative paradigm for ASR started in end of1980s. After decades of research and many successfully deployed commercial products, the performance of automatic speech recognition is not up to par with human speech recognition.

In the paper the author proposes a new model a hybrid HMM and Deep Neural Networks. These networks can be train using the embedded Viterbi algorithm. CD-DNN-HMMs outperform CD-GMM-HMMs in terms of recognition accuracy on tasks and thus are better than the traditional HMM model. We will empirically compare the performance difference between using a monophone alignment and a tri-phone alignment, using monophone state labels and tri-phone senone labels, using 1.5 K and 2 K hidden units in each layer, using an ANN-HMM and a DNNHMM, and tuning and not tuning the transition probabilities. With the experiments performed using both the model we can observe that DNN-HMM model(the hybrid model) proposed by authors of the paper has outperformed traditional HMM model in every respect.

Problem 5

Part A

We can create a genetic algorithm that classifies people as 'accepted to an Ivy League School' by taking the various portions of the Ivy League training data and running the parameters through a decision tree that will eventually classify students as Ivy League 'accepted' or not. The decision trees need to use each of the event parameters to decide Ivy League acceptability. We can encode this decision trees as strings using the method we discussed in class. The root of the tree will be the first element in the string. Assuming the string uses zero indexing, the children of the root (index zero) would be at indices 1 and 2. So given the root node, we can find the string position of the root's children. Likewise using this similar form of encoding the string, we can find any node's child's location by taking the node's index and performing the following operation, 2(i + 1) and 2(i + 1) + 1, for left and right children respectively. Thus, we can represent a decision tree as a flattened string, where the children of your current position in the tree (i) are represented by the equation I described above.

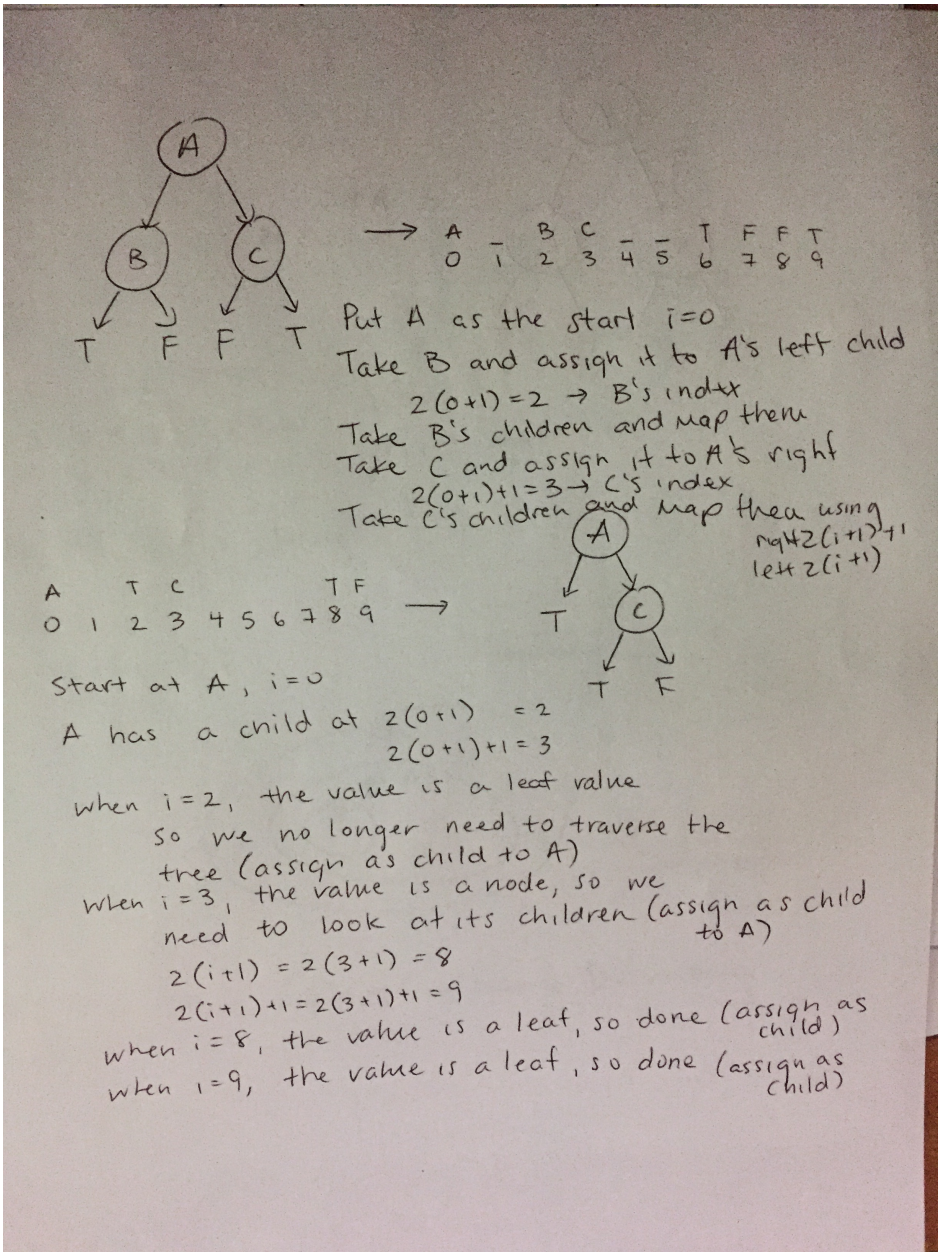
It is important to note that each node is a decision that is either true or false, which leads to another part of the tree.

To make sure it's clear I've illustrated an example of covering a simple tree of height two to a string in the figure below. The figure below contains an example for encoding and decoding the string.

Part B

To map the encoding into a decision tree, we can start at the root node and recursively or iteratively build the tree by traversing the string. As described before, the children of a given node are given by the product of two and the sum of the index and 1 or 2(i+1) for the left node and 2(i+1) + 1 for the right node. So starting at the root, we can find all nodes connected to the tree by traversing it in this manner and building a tree as we go.

To illustrate how the mapping works better, I have included an example below to make sure it's clear how we can map our string back to a decision tree.



Part C

The hypothesis space enabled by this encoding is all boolean decision trees. This is due to the constant factor of 2. Hypotheses that cannot be easily represented are those with a larger branching factor than 2. This is because of the constant factor in the mapping function of 2. If we changed this constant factor, our encoding could represent a larger hypothesis space. Yes the full set of Boolean functions is representable by this encoding for the data.

Part D

The possible decision trees that can be represented by ID3 is greater than those that can be represented by the encoding we have come up with. In other words, the hypothesis space representable by ID3 is greater and contains the hypothesis space representable by our boolean decision tree encoding. This is because ID3 has a larger branching factor and does not rely so heavily on boolean functions. Additionally, the decision one makes at a given point in an ID3 decision tree is influenced by probability, which is not encoded in our genotype. We could modify our encoding to represent a greater hypothesis space by changing how decisions are made at each node and by increasing the branching factor to n branches, where n is the number of decisions that can be made at any given node.

Part E

The easiest fitness function for a string encoded decision tree is to decode and map the string back into a tree so that we can test the tree on a sample or a large portion of our training data to determine if the tree classifies properly. We can make our fitness function the error of classification. So the lower the error, the more fit a decision tree is. So one simple way to ensure that random sampling doesn't result in error is we can run all the data points through the decision tree and determine if the percent error in classification. While this would be slow it would prevent sampling error. Alternatively, as long as you have a large enough sample size that is representative of the data, you could classify on a subset of the data.

Part F

We can define how mutation works by swapping nodes in the tree and in turn modifying the tree significantly. In terms of string manipulation this would involve keeping track of the connectivity of the tree and would be slow. An alternative way is given a tree, it might be possible to simply bit flip the leaf nodes, which are the only results that matter in the trees classification. Swapping would work very much like this bit flipping, but it would allow for different trees with different decisions being made at each node step.

So when swapping nodes, we need to keep track of all the nodes children and all of their children and so on. And using this we can perform a swap, which would involve trading children to preserve the rest of the resulting tree. And modify only the decision that gets made through the swap.

Bit flipping can also work by only flipping the true false values of leaf nodes to see if this classifies better or worse. To find the leaf nodes, we could just traverse the tree and find a node that has leaf nodes and if they are leaf nodes, flip/swap the values of the left and right. For a node with one leaf node and another node, you define the same rule flip the leaf and flip the node's value.

Either of these methods of mutation would only be performed once per mutation cycle. And choosing where to swap or flip would be a random process that would choose sections further down in the tree. This could be accomplished by using the $\log_2$(string length) to determine how far down in the tree to perform the split or swap, giving preference to the second half.

Part G

The crossover would work by splitting at one random point in the string. This would likely result in a few weaker and a few stronger mutations. A single point is better because it becomes difficult to concatenate strings that represent trees that actually makes sense. By splitting in one

section, we may get some trees that are shorter than others that might not represent the entire classification space. Some trees might be longer than others because by taking a random place on the tree, the later half of the information may become useless if there is no pointer to that section of the string. However, it is more likely that a full tree would classify better than a short one so these would likely not last long due to life span aspect of the algorithm

Part H
The selection process would be fitness proportional and so those with a high fitness would cross with those with high fitness. This selection process would hopefully mitigate issues with crossover because we anticipate that trees that represent the entire space would be better than those that do not. This selection would probably be thresholded. So the top 50% would cross with the top 50% and the bottom 50% would cross with the bottom 50%. This enables an element of randomness to still exist, which is necessary because of the representation of the tree as a string.

Part I
Given that the population might consist of short trees that do not classify as well as their parents, I think that is very necessary to have parents survive past their children. We would determine lifespan for the population based on the fitness and threshold it. So only the top 90% of the population survive past the next round of the algorithm.

Part J
For the termination condition, I think it's best to use a fixed number of generations for a stepwise increasing population fitness. So provided that the number of generations doesn't exceed some number, then we can increase the fitness we expect out of the population. If the fitness of the population does not exceed some threshold and the number of generations limit for that threshold is exceed we will terminate. In this way, if the algorithm doesn't work well, but can converge to a solution, we can still run the code and achieve a decision tree that works moderately well.